HAL
open science

Defining Component Protocols with Service Composition: Illustration with the Kmelia Model

Pascal Andre, Gilles Ardourel, Christian Attiogbé

To cite this version:

Pascal Andre, Gilles Ardourel, Christian Attiogbé. Defining Component Protocols with Service Composition: Illustration with the Kmelia Model. Software Composition, 6th International Symposium, SC 2007, Mar 2007, Braga, Portugal. pp.2-17, 10.1007/978-3-540-77351-1 . hal-00397713

HAL Id: hal-00397713

https://hal.science/hal-00397713

Submitted on 23 Jun 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Defining Component Protocols with Service Composition: Illustration with the Kmelia Model

Pascal André, Gilles Ardourel, Christian Attiogbé

LINA CNRS FRE 2729 - University of Nantes
F-44322 Nantes Cedex, France
(Pascal.Andre,Gilles.Ardourel,Christian.Attiogbe)@univ-nantes.fr

Abstract. We address in this article the description and usage of component protocols viewed as specific services. In addition to inter-component service composition, our Kmelia component model supports vertical structuring mechanisms that allow service composition inside a component. The structuring mechanisms (namely state annotation and transition annotation) are then used to describe protocols which are considered here as component usage guides. These structuring mechanisms are integrated in the support language of our component model and are implemented in our COSTO toolbox. We show how protocol analysis is performed in order to detect some inconsistencies that may be introduced by the component designers.

Key words: Component, Service, Composition, Protocols, Property Analysis

1 Introduction

In this work we address the description and usage of component protocols viewed as specific services and described as such. In [9] Meyer suggests a property classification for a Component Quality Model that may lead to trusted components. We consider the *assertions* and *usage documentation* properties which range in the *Behaviour* category from the classification. The first property requires formal descriptions which are helpful to ensure the correctness of the components and their assemblies. The *usage documentation* property requires specific abstraction means in order to help the component-based system developer to build correct assemblies. Clearly, this component documentation property participates in the development of trusted components: this motivates our work. In this context, component documentation should therefore be more than a list of available services (like IDL descriptions); it should overview the component behaviour and constraints, provide some guidelines to use services, describe precisely the usage conditions of services and the interaction conditions. These requirements are fulfilled by the present work which builds on the Kmelia component model [4] which is an abstract component model based on services. Kmelia services are more than simple operations: they enable complex interactions and are the key

element to model components and to connect them to make assemblies. The use of service is central to the verification of compatibility when assembling components according to four compatibility layers: signature, structure, contracts and behaviours layers. In a previous article [4] we presented the Kmelia model and we studied the definition and the verification of component assemblies which are based on a *horizontal service composition*. In the present article we extend the service composition.
In the horizontal composition, services of the same level in various components are composed, with respect to the four compatibility levels, to define new services.

To enforce the idea of component documentation, we consider a methodological layer between services and components. This layer deals with the good usage of the components: which services can be used to fulfil a given need and in what order these services should be called. This layer corresponds to the concept of *component protocol* already used in various component models. Compared with related approaches (see Section 4) which are provider-oriented protocols, our proposal suggests user-oriented protocols. This means that the Kmelia component protocols are not a component life-cycle or a component constraint but merely *macro-services* which play an important role in component composition. To support protocols in Kmelia we now introduce a *vertical service composition*, based on hierarchical structuring operators, to build new provided services from existing ones. Building protocols with service composition is beneficial because: the component model stays simple; protocols can be combined and can play a central role in component composition and last, the verification support of service composition may be reused.

The contribution of this article is twofold: new vertical service composition operators are introduced with their formal descriptions; the definition of powerful component protocols, using service composition, to structure the component interface. From the verification point of view we reuse the existing techniques developed for the service level and we adapt them to the protocol level.

The article is structured as follows. Section 2 is a brief overview of the Kmelia formal component model. In Section 3 we define the vertical service composition. Component protocols are developed in Section 4; first we discuss the concept and compare it with related approaches; then we define protocols in Kmelia and illustrate with an example of a bank Automatic Teller Machine system. The verification aspect is studied in Section 5. Last, we conclude in Section 6 and discuss some perspectives.

2 Overview of the Kmelia Component Model

Kmelia is a component model based on services [4]: an elementary Kmelia component encapsulates several services (Fig. 1). The service behaviours are captured with labelled transition systems. Kmelia makes it possible to specify abstract components, to compose them and to check various properties. A Kmelia abstract component is a mathematical model of an open multi-service system that supports synchronous communication with its environment. A component

specification language (also named Kmelia) and a prototype toolbox (COSTO) support the Kmelia model. The toolbox already permits formal analysis via Lotos/CADP[1] and Mec[2]. We recall (from [4]) in the following the main definitions and the related notations to facilitate the reading of the article.

```
Component C1
 Interface   <Interface descr>
 Types       <Type Defs>
 Variables   <Var list>
 Invariant
             <Predicate>
 Initialisation
  ... // var. assignments

 Services
  ... // as described at side
end
```

```
Provided aService_1 ()
 Interface  <Interface descr>
 Pre        <Predicate>
 Post       <Predicate>
 Behaviour
 init       aStateI
 final      aStateF
 { state_i --label--> state_j
   ... }
end
Required aService_2 ()
 ... //in the same way
```

Fig. 1. Overview of Kmelia syntax

Service Description A *service* s of a component C is defined with an *interface* I_s and a (dynamic) *behaviour* $\mathcal{B}_s$: $\langle I_s, \mathcal{B}_s \rangle$. The interface I_s of a service s is defined by a 5-tuple $\langle \sigma,\ P,\ Q,\ V_s, S_s \rangle$ where σ is the service signature (name, arguments, result), P is a precondition, Q is a postcondition, V_s is a set of local declarations and the *service dependency* S_s is a 4-tuple $S_s = \langle sub_s,\ cal_s,\ req_s,\ int_s \rangle$ of disjoint sets where sub_s (resp. cal_s, req_s, int_s) contains the provided services names (resp. the services required from the caller, the services required from any component, the internal services) in the s scope.

The behaviour $\mathcal{B}_s$ of a service s is an *extended labelled transition system* (eLTS) defined by a 6-tuple $\langle S, L, \delta, S_0, S_F, \Phi \rangle$ with S the set of the states of s; L is the set of transition labels and δ is the transition relation ($\delta \in S \times L \rightarrow S$). S_0 is the initial state ($S_0 \in S$), S_F is the finite set of final states ($S_F \subseteq S$), Φ is a *state annotation* relation ($\Phi \in S \leftrightarrow sub_s$). The transitions in δ (with the$((ss, lbl),\ ts)$ abstract form) have the `ss--lbl-->ts` concrete form.
The transition labels are (possibly guarded) combinations of actions: `[guard] action*`. The actions may be either *elementary actions* or *communication actions*. An elementary action (an assignment for example) does not involve other services; it does not use a communication channel. A communication action is either a *service call/response* or a message *communication*.

[1] www.inrialpes.fr/vasy
[2] altarica.labri.fr

Component Description A component C is a 8-tuple $\langle \mathcal{W}, Init, \mathcal{A}, \mathcal{N}, I, \mathcal{D}_S, \nu, \mathcal{C}_S \rangle$ with:

- $\mathcal{W} = \langle T, V, V_T, Inv \rangle$ the state space where T is a set of types, V a set of variables, $V_T \subseteq V \times T$ a set of typed variables, and Inv is the state invariant;
- $Init$ the initialisation of the V_T variables;
- $\mathcal{A}$ a finite set of elementary actions;
- $\mathcal{N}$ a finite set of service names;
- I the component interface which is the union of two disjoint finite sets: I_p the set of names of the provided services and I_r the names of required services.
- $\mathcal{D}_S$ is the set of service descriptions which is partitioned into the provided services ($\mathcal{D}_{S_p}$) and the required services ($\mathcal{D}_{S_r}$).
- $\nu : \mathcal{N} \rightarrow \mathcal{D}_S$ is the function that maps service names to service descriptions. Moreover there is a projection of the I partition on its image by ν: $n \in I_p \Rightarrow \nu(n) \in \mathcal{D}_{S_p} \wedge n \in I_r \Rightarrow \nu(n) \in \mathcal{D}_{S_r}$
- $\mathcal{C}_S$ is a constraint related to the services of the interface of C in order to control the usage of the services.

The component behaviour relies on the behaviours of its services. The Kmelia components are composable via the interfaces of the involved services. Interface-compatible and behaviour-compatible services are composed at various levels to build *assemblies*. Assemblies and services can be encapsulated into a larger component called a *composition*.

3 Service Composition

In this section we consider two dimensions for service composition; each dimension is related to service behaviour (eLTS). The first dimension already presented in [4] deals with horizontal structuring mechanisms to compose services and components from existing ones on the basis of a client-supplier relation. The second dimension is introduced in this article; it deals with vertical structuring mechanisms for building new services.

3.1 Horizontal Structuring Mechanisms

Horizontal service composition is tightly coupled with component composition and hierarchical links between components. The horizontal structuring mechanisms are established by linking required services to services which are provided either internally or by the caller service or by a third component. These service calls are handled with communication mechanisms. The services are described in such a way that their interactions are made explicit via communication mechanisms. We use communication channels and the standard communication primitives `!` and `?`; they are complemented with `!!` and `??` to deal respectively with service call and service wait. Indeed as service interactions are not elementary, we distinguish their communication operators from the primitive ones.
The interacting services are viewed (from an observer) as one service. Inter-component interactions are based on service behaviour communications. The communications that support the interaction and hence the composition, are

matching pairs: *send message(!)-receive message(?), call service(!!)-wait service start(??), emit service result(!!)-wait service result(??).*
Two services are composable if their signatures are matching (types), the assertions are consistent, the (hierarchical) service dependencies are not conflicting and their behaviours are compatible. When services are composed, they are linked via the information available in their interfaces. Provided services are linked to corresponding required services. In the same way, subservices are linked between the composed services. The transition labels of the service behaviours are used to perform the running of the resulting behaviour: either we have independent behaviours or a synchronising behaviour in the case of mathing labels.

3.2 Vertical Structuring Mechanisms

In the following we consider and formalise two *vertical* structuring mechanisms that enable us to structure hierarchically the services: they are the *state annotation* mechanism and the *transition annotation* mechanism. Additionally to the flexibility of service description with *optional behaviours* (syntactically expressed as a state annotation) or *mandatory behaviours* (syntactically expressed as a transition annotation) the structuring mechanisms provide a means to reduce the LTS size, to share common services or subservices and to master the complexity of service specification, while preserving the pre/post condition contract at the begining/termination of services (both client and supplier constraints).

We maintain the principle that formally the unfolding of an eLTS should result in a LTS (in a recursive way). The unfolding of a service consists in the unfolding of all its annotated states ($state_unfold$ in the sequel) and the unfolding of the annotated transitions ($transition_unfold$ in the sequel). For the formalisation we use the (standard) operational semantics rules with premises and consequences separated by an horizontal line.

The `<< >>` structuring operator We use the `<< >>` operator to denote an optional service call at any state of a service running. The principle is that the caller of a service s, of a component C, may call a service ss that belongs to the provided interface sub_s of s, when the running of s reaches a state e_i (of the LTS of s) annotated with ss.
This *optional* service call is syntactically noted with $\mathtt{e}_i$ `<<ss>>` in the eLTS of s. In [4] the state annotation mechanisms (called *branching states*) was informally introduced. According to the established link between a required and a provided service, there is a renaming which results in a uniform link name. Therefore, the service call is performed with `_linkName!!serviceName(...)` where `_linkName` (resp. `serviceName`) stands for the established link name (resp. the service name).
Let us illustrate with the example in the Figure 2[3]. It represents the main ser-

[3] This picture is generated by the KmeliaToDot module of our COSTO toolbox.

vice of the user interface component of a bank ATM specification[4]. This service asks either for a withdrawal (`_ask_for_money!!ask_for_money`) or for a query account (`_query_account!!query_account`). The `e1`, `e2` and `e10` states are annotated with `<<code>>`; it means that the `code` service can be called from this state by the service which is interacting with the current one.

Fig. 2. An example of optional services in the `USER_INTERFACE` component

The relation $\Phi : S \leftrightarrow sub_s$ is used to manage the annotated states of a service specification (see Section 2). Now let formalise the structuring mechanisms introduced via state annotations. Let s be a service, e_j and e_i (annotated with ss) be two states of s. Let ss, a member of sub_s, be a service provided by (the interface of) s. The behaviour of a service ss is also an (extended) labelled transition system defined by a 6-tuple $\langle S_{ss}, L_{ss}, \delta_{ss}, \Phi_{ss}, S_{0_{ss}}, S_{F_{ss}} \rangle$.

The semantics of the unfolding of annotated states (in the domain of Φ) is as follows. We use the standard α-conversion to rename states and transitions to avoid name conflict. For this purpose, α_{state_s} denotes a renaming function that renames its parameters so as to avoid conflicts with the state names in s. The $\alpha_{transition_s}$ and α_{label_s} functions are used in the same way to denote transition and label renaming.

$$\frac{\begin{array}{c} s \mathrel{\widehat{=}} \langle S_s, L_s, \delta_s, \Phi_s, S_{0_s}, S_{F_s} \rangle \quad \wedge (e_i, ss) \in \Phi_s \quad \wedge \\ ss \mathrel{\widehat{=}} \langle S_{ss}, L_{ss}, \delta_{ss}, \Phi_{ss}, S_{0_{ss}}, S_{F_{ss}} \rangle \quad \wedge \\ S^{\alpha}_{ss} = \alpha_{state_s}(S_{ss}) \quad \wedge L^{\alpha}_{ss} = \alpha_{label_s}(L_{ss}) \quad \wedge \delta^{\alpha}_{ss} = \alpha_{transition_s}(\delta_{ss}) \quad \wedge \\ ss_{\alpha} \mathrel{\widehat{=}} \langle S^{\alpha}_{ss}, L^{\alpha}_{ss}, \delta^{\alpha}_{ss}, \Phi_{ss}, S^{\alpha}_{0_{ss}}, S^{\alpha}_{F_{ss}} \rangle \quad \wedge \\ S'_s = S_s \cup S^{\alpha}_{ss} \wedge L'_s = L_s \cup L^{\alpha}_{ss} \cup \{\texttt{??}\ \texttt{ss}\} \quad \wedge \\ \delta'_s = \delta_s \cup \delta^{\alpha}_{ss} \ \cup \ \{((e_i, \texttt{??}\ \texttt{ss}), S^{\alpha}_{0_{ss}})\} \ \cup_{S_{f_{ss}} \in S^{\alpha}_{F_{ss}}} \{((S_{f_{ss}}, \epsilon), e_i)\} \quad \wedge \\ \Phi'_s = \Phi_s - \{(e_i, ee)\} \quad \wedge \\ S'_{0_s} = S_{0_s} \wedge S'_{F_s} = S_{F_s} \end{array}}{state_unfold(s, ee) = \langle S'_s, L'_s, \delta'_s, \Phi'_s, S'_{0_s}, S'_{F_s} \rangle}$$

[4] This ATM specification deals with the interaction between component services in order to enable some functionalities provided by the ATM: withdrawal, query account, etc. Some of these functionalities need the code or the amount from the user [4].

The rule expresses that after the unfolding of e_i, a transition labelled with `??ss` goes from the annotated state to the initial state of the ss service; if there is a call to the service ss from the e_i state, provided that the precondition of ss is true, this transition (as the other matching action) will lead to the initial state of ss. To handle the end of the ss service, where the postcondition of ss is true, a transition labelled with ϵ relates the final states of ss and the annotated state; finally, all the transitions of ss are allowed in s provided that the control reaches ss (hence the inclusion of transition relations).

The `[[ ]]` structuring operator The `[[ ]]` operator denotes *mandatory* service calls at any stage of a service running. To follow a transition annotated with `[[ss]]` the caller of a service s must call the service ss that belongs to the provided interface sub_s of s. Again pre/postcondition contract is preserved. Only one service name is allowed for this operator. In the same way as for state annotation, we extend the LTS of the service behaviour with a relation $\Psi : S \times S \leftrightarrow sub_s$ to capture the annotated transitions. Note that to preserve the service composition techniques and existing tools we do not modify the δ relation.
We use three components to describe the ATM example: the `USER_INTERFACE` component which provides the `behaviour` service and requires the `amount` service; the `ATM_CORE` component which provides the `withdrawal` service and requires the `ask_amount` service and the `ATM_BASE` component. The service `withdrawal` is linked with the `ask_for_money` one; the link name is `_ask_for_money`. In the same way the `ask_amount` service is linked with `amount` resulting in the `ask_amount` link. As depicted in the Figure 3, the `amount` service of the `USER_INTERFACE` must be called (here from the `withdrawal` service) after the `b2` state.

Fig. 3. An example of mandatory service in the `USER_INTERFACE` component

In the same way as for the operator `<< >>` we give the semantics of the `[[ ]]` operator. Consider a transition between e_i and e_j which is annotated with ss: we have $((e_i, e_j), ss)$ in Ψ. The semantics of the unfolding of the transition is as follows.

$$
\begin{array}{c}
s \mathrel{\widehat{=}} \langle S_s, L_s, \delta_s, \Phi_s, S_{0_s}, S_{F_s} \rangle \quad \wedge \\
((e_i, ss), e_j) \in \delta_s \quad \wedge (e_i, e_j) \in \mathrm{dom}(\Psi) \quad \wedge \quad \Psi(e_i, e_j) = ss \wedge \\
ss \mathrel{\widehat{=}} \langle S_{ss}, L_{ss}, \delta_{ss}, \Phi_{ss}, S_{0_{ss}}, S_{F_{ss}} \rangle \quad \wedge \\
S^{\alpha}_{ss} = \alpha_{state_s}(S_{ss}) \quad \wedge L^{\alpha}_{ss} = \alpha_{label_s}(S_{ss}) \quad \wedge \delta^{\alpha}_{ss} = \alpha_{transition_s}(\delta_{ss}) \quad \wedge \\
ss_{\alpha} \mathrel{\widehat{=}} \langle S^{\alpha}_{ss}, L^{\alpha}_{ss}, \delta^{\alpha}_{ss}, \Phi_{ss}, S^{\alpha}_{0_{ss}}, S^{\alpha}_{F_{ss}} \rangle \quad \wedge \\
S'_s = S_s \cup S^{\alpha}_{ss} \wedge L'_s = L_s \cup L^{\alpha}_{ss} \cup \{\texttt{?? ss}\} \quad \wedge \\
\delta'_s = \delta_s \cup \delta^{\alpha}_{ss} - \{((e_i, ss), e_j)\} \\
\cup \{((e_i, \texttt{?? ss}), S^{\alpha}_{0_{ss}})\} \cup_{S_{f_{ss}} \in S^{\alpha}_{F_{ss}}} \{((S_{f_{ss}}, \epsilon), e_j)\} \quad \wedge \\
\Psi'_s = \Psi_s - \{((e_i, ee), e_j)\} \quad \wedge \\
S'_{0_s} = S_{0_s} \wedge S'_{F_s} = S_{F_s} \\
\hline
transition_unfold(s, t_i) = \langle S'_s, L'_s, \delta'_s, \Phi'_s, S'_{0_s}, S'_{F_s} \rangle
\end{array}
$$

The semantic rule expresses that when a transition annotated with ss exists between the states e_i and e_j, then an expansion of the ss service is performed between e_i and e_j. The behaviour of ss is then reachable from the e_i state via a wait of a call (`??ss`) ensuring the precondition of ss; after the running of ss (one reaches a final state), the postcondition of ss is established and the execution proceeds from the e_j state due to the ϵ transition. A side effect is considered here; the Ψ relation that extends the service specification is also updated along the semantic rule. This rule is sufficient to deal with all annotation cases. The various cases of transition annotation are dealt with as follows:

- when an annotated transition is guarded $(((e_i, \texttt{[g] [[ss]]}), e_j) \in \delta)$, the firing of the transition depends on the value of the guard; in this case the semantics rule is slightly changed as follows;

$$
\begin{array}{c}
s \mathrel{\widehat{=}} \langle S_s, L_s, \delta_s, \Phi_s, S_{0_s}, S_{F_s} \rangle \quad \wedge \\
((e_i, \texttt{[g] ss}), e_j) \in \delta_s \quad \wedge (e_i, e_j) \in \mathrm{dom}(\Psi) \quad \wedge \quad \Psi(e_i, e_j) = ss \wedge \\
ss \mathrel{\widehat{=}} \langle S_{ss}, L_{ss}, \delta_{ss}, \Phi_{ss}, S_{0_{ss}}, S_{F_{ss}} \rangle \quad \wedge \\
S^{\alpha}_{ss} = \alpha_{state_s}(S_{ss}) \quad \wedge L^{\alpha}_{ss} = \alpha_{label_s}(L_{ss}) \quad \wedge \delta^{\alpha}_{ss} = \alpha_{transition_s}(\delta_{ss}) \quad \wedge \\
ss_{\alpha} \mathrel{\widehat{=}} \langle S^{\alpha}_{ss}, L^{\alpha}_{ss}, \delta^{\alpha}_{ss}, \Phi_{ss}, S^{\alpha}_{0_{ss}}, S^{\alpha}_{F_{ss}} \rangle \quad \wedge \\
S'_s = S_s \cup S^{\alpha}_{ss} \quad \wedge \quad L'_s = L_s \cup L^{\alpha}_{ss} \cup \{\texttt{?? ss}\} \quad \wedge \\
\delta'_s = \delta_s \cup \delta^{\alpha}_{ss} - \{((e_i, \texttt{[g] ss}), e_j)\} \\
\cup \{((e_i, \texttt{?? ss}), S^{\alpha}_{0_{ss}})\} \cup_{S_{f_{ss}} \in S^{\alpha}_{F_{ss}}} \{((S_{f_{ss}}, \epsilon), e_j)\} \quad \wedge \\
\Psi'_s = \Psi_s - \{((e_i, ee), e_j)\} \quad \wedge \\
S'_{0_s} = S_{0_s} \wedge S'_{F_s} = S_{F_s} \\
\hline
unfold_gtransition(s, t_i) = \langle S'_s, L'_s, \delta'_s, \Phi'_s, S'_{0_s}, S'_{F_s} \rangle
\end{array}
$$

- when an annotated transition is one of the output transition of a node (there is a choice of transitions), the used transition is the one which is involved in the current interaction with another service that call (or which is called by) the current one.

3.3 Component Maintenance and Consistency

Component maintenance Decomposing a large behaviour into subservices is encouraged in Kmelia, but it bears consequences if the service was already used by

other services. For instance, when the behaviour of an existing component service s is modified using the `[[]]` operator to exploit a part of it as a new service ss, the existing clients of s will cease to be compatible because they miss the (new) connection to ss. Indeed, the use of `[[]]` to modify s creates new transitions between s and ss: especially a call to ss which is of course not included in the previous client of s. This is what we called *interface granularity mismatch* in [2]: a client service considers that all the communications are made in the context of the unique old service while other newer clients use the new subservice ss. While being quite difficult to address in the general case, the granularity mismatch is easily avoided in the case of a maintenance or refactoring operation. For this reason we use a rather flexible operator noted `[||]` which expands in the same way as the (inflexible) `[[]]` operator but which adds new transitions that allow old clients to circumvent both the call to the subservice and the waiting for its termination. Likewise the flexible counterpart of (the inflexible) `<< >>` is the `<| |>` operator.

Formally the flexible operators have rules very similar to their inflexible counterparts. We do not detail them here; the main point is that in the case of `<| |>` and `[||]` the final states of ss may not be reached, therefore an ϵ-transition relates each predecessor of these final states to e_j. Indeed the new clients call and wait for the termination, but the existing clients do not. In the case of this formalisation δ'_s is changed as follows:

$$\delta'_s = \delta_s \ \cup \ \delta^{\alpha}_{ss} \ \cup \{((e_i, \epsilon), S^{\alpha}_{0_{ss}})\} \ \cup_{e_p \in \{q_p | ((q_p, lx), s_{f_{ss}}) \in \delta_{ss}\}} \{((e_p, \epsilon), e_j)\}$$

Thanks to the flexible versions of the vertical structuring mechanisms, decomposing large services into subservices is expected to be a common refactoring. The systematic detection of occurrences where such refactorings are performed will be needed; but the adaptation of subservices that use parameters are out of the scope of this paper.

Impact of structuring on service consistency The previous structuring mechanisms are independent of the service behaviour but they can impact on its consistency. The correct ordering of services may be checked using preconditions and postconditions. Therefore some control may already be performed at the provider side. We study these problems and provide some solutions in the following in the specific case of the component protocols.

Now we have a component model entirely equipped with service structuring mechanisms. The added vertical structuring mechanisms do not impact on composition since they are defined in terms of elementary LTS. However it is necessary to check for possible design errors. In the following section we reuse the service composition mechanisms to describe component protocols.

4 Component Protocols

Component behaviour protocols [14, 12, 8] have been introduced to extend static component interfaces to dynamic constraints such as valid sequences of message exchange, valid condition of service invocation, connection handling, etc.

4.1 The Component Protocol Concept

The concept of protocol already exists in several component or service models but its meaning varies from one model to another. In some approaches a protocol is a specific layer in a contractual vision including assertions [5, 6, 4, 10] and non-functional constraints like the quality of service [5, 6]. In other approaches [1, 5, 7, 14] protocols are communication rules on connectors where adaptation is possible. Protocols can also be recursive [13, 15] or subtyped [5, 12, 14].

In a short comparison[5], we use four criteria to compare the approaches: (1) contents of the protocols (service invocation, actions, message exchange, control structures...), (2) the attachment unit (component, interface, service, connector or architecture), (3) the formalism itself (finite state machine, statecharts, regular expressions, etc), (4) property specification and proof support techniques (temporal logic, markup language, algorithms, etc). We hereby classify these related approaches into three categories where the attachment unit is the main criteria:

1. The first category groups the approaches which define a protocol as a component *lifecycle* [5, 8, 11, 13, 15]. A single protocol is associated to the component (or with its single interface). The component is a process and the services are either atomic (messages) or defined by a specific behaviour [10].
2. In the second category a protocol defines a component view's lifecycle. In some of these approaches, a protocol is associated to an interface and several interfaces coexist in the component [3, 6, 12]. In other approaches [1, 7, 14] a protocol handles the communications on connection points (just like a usual communication protocol).
3. In the third category [4] a protocol describes a particular use of the component. Several protocols coexist within the component in one or several interfaces.

The above approaches are not different in terms of expression power but they are in terms of abstractions (concepts) from the component client point of view. For example, using a basic component model (single interface, single protocol), one can model every component system and in particular a system where connectors are considered as components and multiple interfaces as component compositions. In such a case the system architect should encapsulate the protocols in composite components and manage the interface consistency (close to the inheritance problems in Object-Oriented Design); this solution leads to heavy modelling. In other words, the approaches of category 1 and some of category 2 consider the protocol as a constraint rather than a guideline for the client. In Kmelia (third category) we rather emphasise the user point of view; this is more developed in the following section.

Protocols as Component Macro-services When a component model does not have the protocol concept, any service of a component can be invoked at any time. This is acceptable for libraries of functions but not for components

[5] available at lina.atlanstic.net/fr/equipes/team10/Kmelia/

whose behaviour evolves with their service behaviours. Indeed the other solutions would be either to use non trivial preconditions for service specifications or to use comments to guide the users. We choose the use of protocol instead.

Component protocols enable the distinction between component state constraints (preconditions), sequencing constraints (ordering) and thereafter make easier the verification of each part. Protocols are both a constraint for the component supplier and a user guide for the component client (*e.g.* use case or scenario):

- A protocol defines the rules which are needed to preserve the component consistency.
- Protocols are helpful for the component system designer in describing guidelines: "which services one can use and in what order one can use them".
- Protocols are a coarse grain for component assemblies: instead of connecting each service, one can connect a pattern of services.

The protocols as considered above, are a means to model user sessions, processes, user classes or communication protocols.

4.2 Specification of Protocols in Kmelia

Within the Kmelia model a component protocol describes a *valid ordering* of service calls. Therefore we beneficially reuse vertical structuring mechanisms to describe protocols; for instance a sequence of mandatory service calls impose an ordering of the services. A protocol stands for a provided service that gives the access to other services of the same component. Thereby a protocol has a behaviour (eLTS). Among the provided services of a component, those used in a protocol description are called *controlled services*; those which are not used in the protocol descriptions are called *free services*. Thereby our model admits the existence of controlled services which are still offered (at any time) through the component interface.

A Kmelia component may provide one or several protocols. The provided protocols may be made *interruptible* by the component designer. The means to do that is the use of a property to qualify some services. Therefore the protocol interfaces have the following form:

```
provided protoName()
         Properties = {protocol, interruptible, ...}
```

A protocol which does not have the *interruptible* property is said *non-interruptible*; once it is started it cannot be interleaved with other runs.

Protocol Specification A protocol p is a specific service; it needs an interface I_p and a behaviour description $\mathcal{B}_p$; therefore we use the same description as for a service: an eLTS. The behaviour of p is specified with $\langle S, L_P, \delta, \Phi, S_0, S_F \rangle$. But to deal with the protocol features, we need some restrictions on the labels of the transitions of protocols. The labels (L_P) are now either annotations (noted `[[ss]]` that corresponds to a service ss which should be called by the service

that uses the protocol) or a local variable manipulation (that corresponds for example to a loop counting or a path predicate).
In the following we adopt the user's point of view, hence using *call to ss* to refer to the annotation of a state or a transition with a service *ss* using the vertical composition operators.

```
provided withdrawProtocol()
 Properties= {protocol,
             nonInterruptible}
 Pre  true
 Post true
 Behaviour
  init  i
  final f
  { i --[[connection]]--> e0
    ...
   }
end
```

Fig. 4. A protocol of the ATM_BASE component

The Figure 4 stands for a component ATM_BASE that includes a protocol `withdrawProtocol`. The protocol gives the user guide of the services `connection`, `withdrawal` and `logout`. This protocol is rather simple, it does not include explicit loops, guards, basic actions on variables, etc. It appears in the component interface in the same way as the other provided services and can be called as such. The services that appear in the protocol (the controlled services) are called in the scope of the protocol in the same way as the subservices of a service are called. As far as the protocol `withdrawProtocol` is concerned, the services `connection`, `withdrawal`, `logout` are controlled but the service `account_query` is free.

5 Formal Analysis and Experimentations

We have undertaken the behavioural compatibility analysis of Kmelia component services [4]. The behaviours of linked services are checked for compatibility: the behavioural analysis is achieved by considering the simultaneous running of two (pairwise) services involved in a communication; the transitions are performed independently if they are labelled with elementary actions; the transitions labelled with communication actions should be matching pairs from both involved services. After the extension of service composition with the vertical structuring mechanisms, the behavioural compatibility analysis of services still works since the new mechanisms do not modify the behavioural structure of our services: we have the (unfolded) LTS of each service labelled with elementary actions or

communication actions. Therefore component interaction *via* composition of services does not change. However, the behavioural compatibility should not hide the general compatibility rules which include assertion checking. The use of the vertical structuring mechanisms may lead to wrong orderings of services (if the user does not pay attention to pre/postconditions). For example, in order to perform *safely* a transition annotated with `[[ss]]` during the execution of a service s, the precondition of ss should be ensured. In the same way, the use of the structuring mechanisms to support protocol description requires a consistency analysis of the protocols. In the following section we investigate one kind of protocol analysis.

5.1 Analysis of Protocols: Inconsistency checking

The absence of inconsistency within protocol descriptions is one of the criteria of a component correctness. For this reason, we need to detect inconsistency in protocols specified by component designers. A protocol of a component is *inconsistent* if one of its service sequences (from the protocol behaviour) is not feasible (*unfeasible sequences*). The following two cases of inconsistency may be detected:

- the existence of guarded sequences of service calls without other choice leading to a final state of the protocol;
- the existence in the protocol of a sequence of service calls $[s_i; s_{i+1}; \cdots ; s_j; s_k]$ such that the post-conditions of s_i to s_j imply the negation of the precondition of s_k; that means, some services called before s_k establish a context which is not altered by other services before the call of s_k and which is not consistent with s_k.
  For instance, if the service `connection` has *not connected* as precondition and *connected* as postcondition then the `connection; connection` sequence leads to an inconsistent protocol (in the same way as any protocol including this sequence).

To analyse and detect unfeasible sequences of service calls, we are experimenting the translation of our needs into properties that will be proved using existing theorem provers such as the Atelier B[6].

5.2 Analysis of Protocols: Inconsistency Detection

This section investigates the inconsistency cases of section 5.1. The goal is to help the component designer to write correct component equipped with protocols. Practically, the analysis of such components will output some warnings or errors showing the wrong parts of the component descriptions. Consider a protocol with its unfolded behaviour and the sub-chains of service calls going from the initial state to a final one (avoiding loops) of the protocol behaviour. For each chain we check for all its sub-chains $s_i; s_j$ (with $j = i + 1$) that

$$\boxed{\neg(post(s_i) \Rightarrow \neg pre(s_j))} \qquad (P1)$$

[6] `www.atelierb.societe.com`

This local property (where $pre(s)$ and $post(s)$ stand for the pre-condition and post-condition of s) should be extended to take into account the effect of a whole chain of calls that precedes a call to a service s_k.
Remind that the eLTS that specifies a protocol behaviour denotes a finite set of sequences which are made of the labels of the transitions. Therefore we have chains made of service calls and simple actions. Practically, a component protocol imposes an ordering of the component running, where each performed service has some effect on the component.
A service (say $s_i = \langle\langle\sigma,\ P_{s_i},\ Q_{s_i},\ V_{s_i}, S_{s_i}\rangle, \mathcal{B}_{s_i})$ is correctly performed if it starts with a state satisfying the required precondition P_{s_i}. $\mathcal{B}_{s_i}$ is the service behaviour; the effect of a service, via its $\mathcal{B}_{s_i}$ behaviour, is indicated by a post-condition Q_{s_i} together with a modification of the component state.
Consequently the initial $(P1)$ property is

$$\boxed{\neg(G_{s_i} \Rightarrow \neg P_{s_j})} \qquad (P2)$$

instead of $\neg(Q_{s_i} \Rightarrow \neg P_{s_j})$ for the chain $s_i; s_j$, where G_{s_i} is a global property. It expresses the cumulative effects of services $s_1..s_i$ on a component just before the call s_j that follows s_i in a chain of the given protocol.

This generalises the situation depicted as follows:

$$\underbrace{s_1}_{G_{s_1}};\ s_2;\ s_3;\ \cdots;\ s_n$$

$$\underbrace{s_1\ ; s_2}_{G_{s_2}};\ s_3;\ \cdots;\ s_n$$

$$\cdots$$

$$\underbrace{s_1;\ s_2;\ s_3;\ \cdots;\ s_i;\ s_j;\ \cdots;\ s_{n-1}}_{G_{s_{n-1}}};\ s_n$$

The predicate P_{s_i} precondition of a service s_i is expressed with local variables (vl_i) that are the parameters of the service and with global variables (vg_k) of the component, together with typing information $(tl_i; tg_k)$ coming from the service and component interfaces:

$$vl_i : tl_i;\ vg_k : tg_k\ .\ P_{s_i}(vl_i, vg_k)$$

In the same way, the predicate P_{s_j} of a service s_j is expressed with local variables (vl_j) of the s_j service and with global variables (vg_k) of the component, together with typing information coming from the s_j service and from the component:

$$vl_j : tl_j;\ vg_k : tg_k\ .\ P_{s_j}(vl_j, vg_k)$$

As we are reasoning independently of the runtime context of the services, the values of local variables are not known (we assume in the best case that they have the right value for the truth of the predicates) when the service are called. The only working hypotheses are those on global variables; therefore we restrict $P_{s_i}(vl_i, vg_k)$ and $P_{s_j}(vl_j, vg_k)$ predicates to $P'_{s_i}(vg_k)$ and $P'_{s_j}(vg_k)$.

The previous property ($P1$)

$$\neg(vl_i : tl_i; vg_k : tg_k \ . \ Q_{s_i}(vl_i, vg_k) \Rightarrow \neg(vl_j : tl_j; vg_k : tg_k \ . \ P(vl_j, vg_k)))$$

is rewritten with

$$\neg(vg_k : tg_k \ . \ Q'_{s_i}(vg_k) \Rightarrow \neg(vg_k : tg_k \ . \ P'(vg_k)))$$

and is generalised with the following *proof obligation*:

$$\boxed{\neg(vg_k : tg_k \ . \ G'_{s_i}(vg_k) \Rightarrow \neg(vg_k : tg_k \ . \ P'(vg_k)))}$$

Finally, detecting inconsistencies results in the systematic checking of this proof obligation on components equipped with protocols. The obligation is yet restrictive (local variables are ignored) but it is possible to alleviate the imposed restrictions; however the obligation proofs will be very complex as we would have to explore some value constraints for local variables. The current compromise (i.e. considering only global variables) helps to detect some inconsistencies with proof obligations which are tractable. Therefore we should integrate, after preprocessing if needed to meet the input language of the prover, the G' and P' predicates with their contexts (types, variables) into the targeted prover. We are using the Atelier B prover as a support for our experimentations.

6 Conclusion and Perspectives

We have extended the horizontal structuring mechanisms of the Kmelia model with two vertical structuring mechanisms: state annotation to deal with optional service calls at some running stage and transition annotation to deal with mandatory service calls when they are needed by the component users. We have shown that these structuring mechanisms, first dedicated to service and component composition, are also appropriate for describing protocols. In this context component protocols are viewed as specific provided services. The behaviour of a protocol is described as a service using a LTS with restricted labels; for example they cannot include basic communication actions. The concept of protocol is added to the model without changing it. The inconsistency of service ordering may be detected through the protocols. Compared to the existing approaches, our abstract component model is easily extensible; it can be incrementally strengthened: in this case by defining the *protocol* property.

We studied protocol inconsistency detection using service pre/post conditions. That led to the generation of obligation proofs that can be managed using existing theorem provers. Robustness with respect to component maintenance was dealt with: when a service is restructured its clients are not broken. We have already implemented the structuring mechanisms within our COSTO toolbox that integrates: Kmelia specification parser, translators to LOTOS and MEC, static interoperability checkers, dynamic interoperability checkers, a translator of Kmelia services into dot (for the visualisation of service behaviours).

The challenge of building trusted components remains exciting. The Kmelia proposal does not yet overcome all aspects of this challenge; additionally to the

improvment of the data and assertion part of the specification language, mechanised correctness analysis of services and components, equipped with protocols or not, are planned as short term research goals. We started some experiments with the Atelier B prover to deal with aspects reated to assertions and not coverered by LOTOS or MEC. In this direction, further work is planned to mechanise the detection of inconsistency. The refinement of Kmelia model into executable framework such as Fractal and SOFA is also an exciting investigation area.

References

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
2. P. André, G. Ardourel, and C. Attiogbé. Coordination and Adaptation for Hierarchical Components and Services. In *Third International ECOOP Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'06)*, pages 15–23, 2006.
3. P. C. Attie and D. H. Lorenz. Establishing Behavioral Compatibility of Software Components without State Explosion. Technical Report NU-CCIS-03-02, College of Computer and Information Science, Northeastern University, 2003.
4. C. Attiogbé, P. André, and G. Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition, SC'06*, volume 4089 of *LNCS*. Springer, 2006.
5. S. Becker, S. Overhage, and R. Reussner. Classifying Software Component Interoperability Errors to Support Component Adaption. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *Proc. of CBSE 2004*, LNCS, pages 68–83. Springer, 2004.
6. A. Beugnard, J-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.
7. C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Adding Roles to CORBA Objects. *IEEE Trans. Softw. Eng.*, 29(3):242–260, 2003.
8. D. Giannakopoulou, J. Kramer, and S-C. Cheung. Behaviour Analysis of Distributed Systems Using the Tracta Approach. *ASE*, 6(1):7–35, 1999.
9. B. Meyer. The Grand Challenge of Trusted Components. In *Proceedings of 25th International Conference on Software Engineering*, pages 660–667. IEEE Computer Society, 2003.
10. OMG. *The OMG Unified Modeling Language Specification, V2.0 Rfp*. Superstructure Specification available at www.omg.org/docs/ptc/05-07-04.pdf, Infrastructure Specification available at www.omg.org/docs/ptc/03-09-15.pdf, 2005.
11. S. Pavel, J. Noye, P. Poizat, and J-C. Royer. Java Implementation of a Component Model with Explicit Symbolic Protocols. In *4th International Symposium on Software Composition, SC'05*, volume 3628 of *LNCS*. Springer, 2005.
12. F. Plasil and S. Visnovsky. Behavior protocols for software components, 2002. IEEE Transactions on SW Engineering, 28 (9), 2002.
13. M. Südholt. A Model of Components with Non-regular Protocols. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Software Composition*, volume 3628 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2005.
14. D.M. Yellin and R.E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.
15. W. Zimmermann and M. Schaarschmidt. Checking of Component Protocols in Component-Based Systems. In *5th International Symposium on Software Composition, SC'06*, volume 4089 of *LNCS*. Springer, 2006.